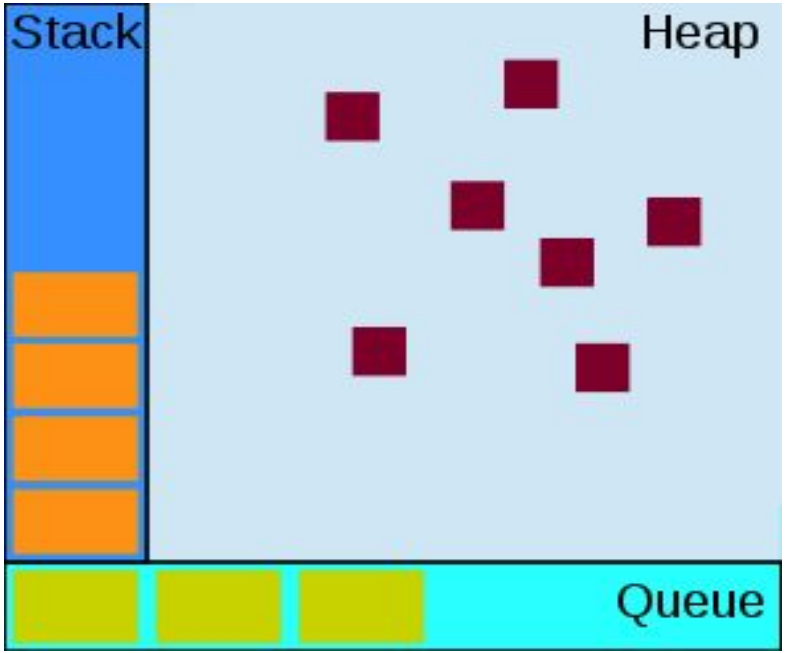

JavaScript

Modelo de Concurrencia I - Event Loop.

JavaScript poseé un modelo de concurrencia basado en un "loop de eventos".

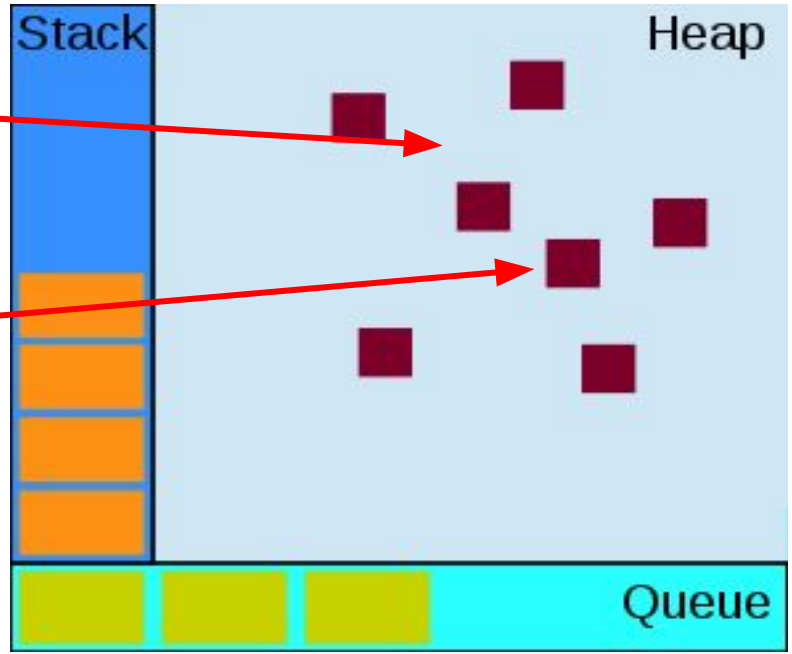
- Single thread.
- No bloqueante.
- Asincrónico.



Modelo de Concurrencia II - Heap.

Denota una gran región de memoria sin estructura ni orden.

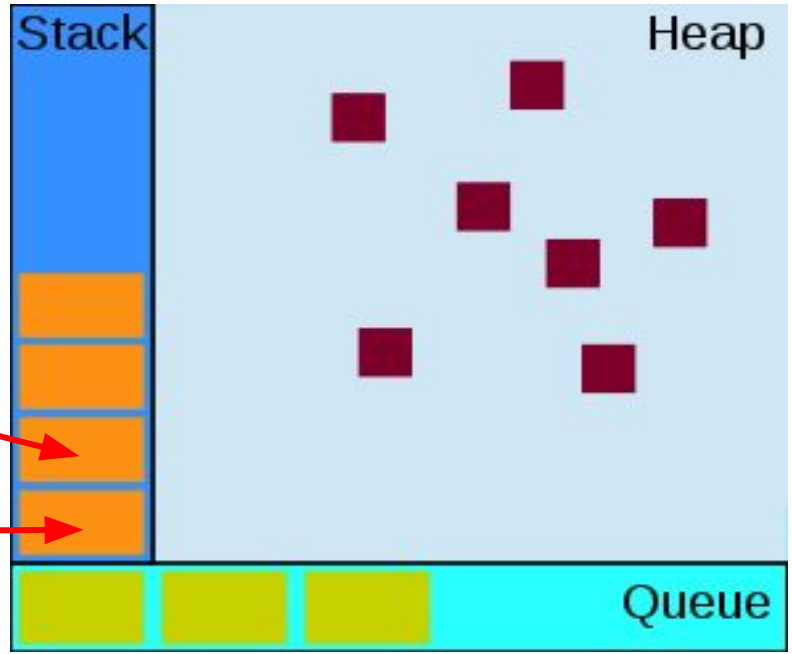
```
var objeto = {
    name: 'Mario'
}


var lista = [1,2,3,4,5,6];
```



Modelo de Concurrencia III - Stack.

Las llamadas a función forman una “pila de frames”.

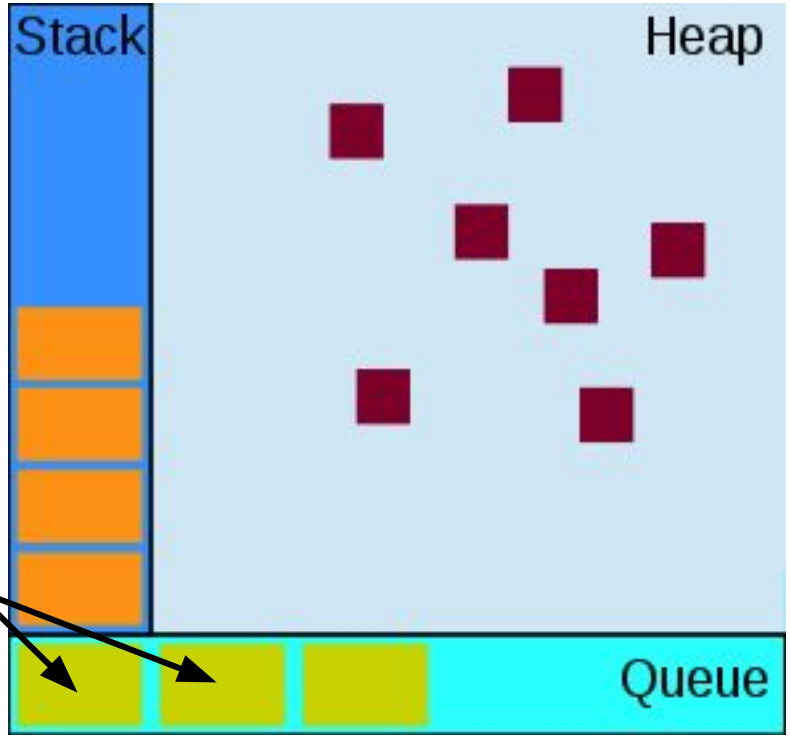
```
var segunda = () =>
console.log('segunda');

var primera = () => {
    segunda();
    console.log('Primera');
}

primera();
```



Modelo de Concurrencia IV - Queue.

```
(function () {
   var mensaje = () => console.log('Función
mensaje');
   console.log('Mensaje inicio');

   setTimeout( () => {
       console.log('Callback 1');
   }, 5000);

   mensaje();

   setTimeout( () => {
       console.log('Callback 2');
   }, 0);

   console.log('Mensaje final');

})();
```

Es una lista de mensajes a ser procesados.



Modelo de Concurrencia IV - Event Loop.

```
while(queue.waitForMessage()){
    queue.processNextMessage();
}
```

Programación Asíncrona

- La programación asíncrona establece la posibilidad de hacer que algunas operaciones devuelvan el control al programa llamante antes de que hayan terminado mientras siguen operando en segundo plano.
- Esto agiliza el proceso de ejecución y en general permite aumentar la escalabilidad, pero complica el razonamiento sobre el programa.

Modelos de programación asíncrona

- Modelo de paso de continuadores (Continuation-passing style).
- Modelo de eventos.
- Modelo de promesas.
- Modelo de generadores.

Continuation-passing style

```
function verificar( clave, callback )
{
    if ( clave != 123 ){
        callback("Clave Errónea");
    }
    else{
        callback(null, "Ok");
    }
}
verificar( 321, (error, exito) => {
    if ( error )
        console.log(error);
    else
        console.log(exito);
})
```

Cada operación no bloqueante recibe una función como último parámetro que incluye la lógica de continuación que debe ser invocada tras la finalización de la misma tanto para procesar los resultados en caso de éxito como para tratar los fallos en caso de error.

Continuation-passing style - Callback Hell

La manera de proceder dentro de este modelo para establecer flujos de ejecución secuenciales exige ir encadenando cada función subsiguiente como continuación de la anterior donde se procesarán los resultados tanto en caso de éxito como de fracaso.

Esto conduce a una diagonalización del código que se ha dado en llamar pirámide del infierno (callback hell), por su falta de manejabilidad práctica en cuanto crece mínimamente el número de encadenamientos secuenciales.

Continuation-passing style - Callback Hell

```
mul(2, 3, function (error, data) {
    if (error)
        console.error (error);
    else{
        div(data, 4, function (error, data) {
            if (error)
                console.error (error);
            else
                add(data, 5, function (error, data) {
                    if ( error )
                        console.error (error);
                    else
                        sum(data, 6, function (error, data) {
                            .....
            });
    });
});
```

Continuation-passing style

Lo bueno: La programación asíncrona basada en continuadores puede ser una buena opción para situaciones en que la lógica de control de flujo es muy sencilla.

- Sencillo para esquemas solicitud / respuesta.
- Alineado con los esquemas de programación funcional.
- Fácil de entender como mecanismo conceptual.

Lo malo: Cuando la lógica de control resulta mínimamente elaborada el proceso de razonamiento sobre el programa se complica lo cual redunda en un código con lógica funcional distribuida y difícil de leer, entender y mantener.

- Complejidad en la definición de una lógica de control de flujo elaborada.
- Difícil establecer mecanismos de sincronización.
- La lógica de control queda distribuida entre cada rama no bloqueante.
- Difícil de seguir, leer y mantener a medida que crece el código.

Modelo de Promesas

Que es una promesa?

- Una promesa es una abstracción computacional que representa un compromiso por parte de la operación no bloqueante invocada de entregar una respuesta al programa llamante cuando se obtenga un resultado tras su finalización.
- La promesa es un objeto que expone dos métodos inyectores then y catch para incluir la lógica de tratamiento en caso de éxito o fracaso.

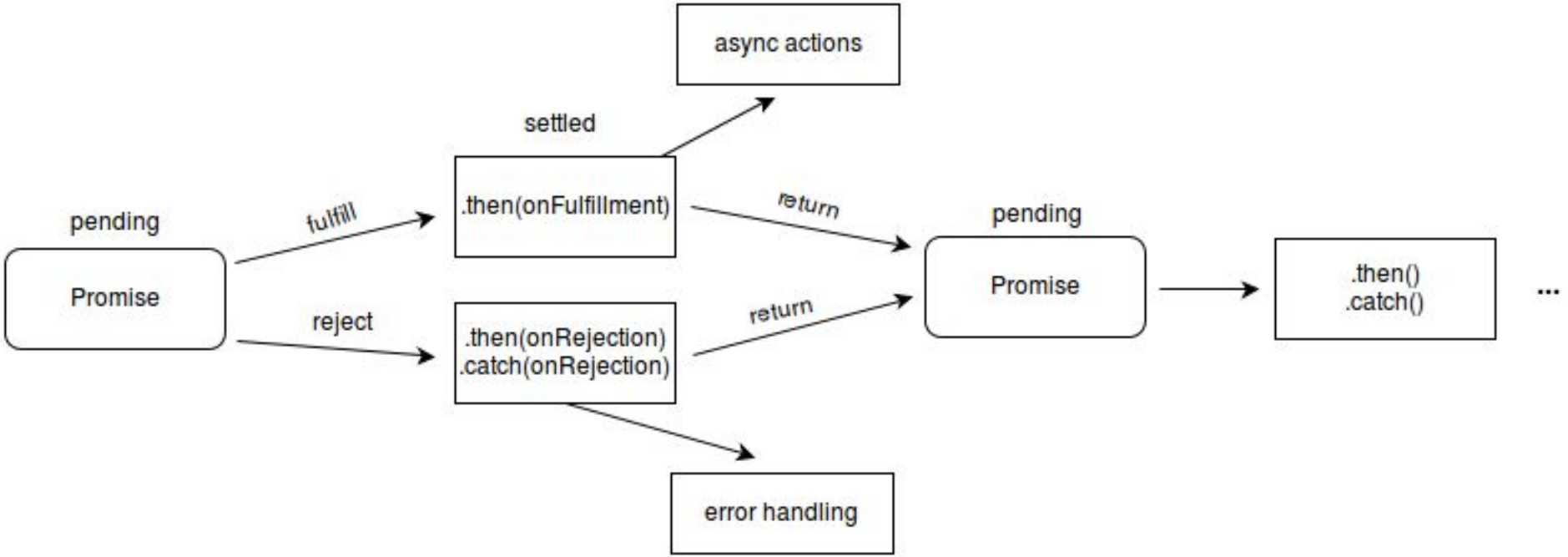
Ciclo de vida de una promesa

- La lógica de tratamiento en caso de éxito o fracaso sólo se ejecuta una vez.
- Se garantiza la ejecución de la lógica de éxito o fracaso, aunque la promesa se resuelva antes de haber inyectado sus manejadores. La promesa espera, si es necesario a disponer de sus manejadores.

Promesas - Estados

Una promesa puede tener uno de los siguientes estados:

pending, fulfilled, rejected.



Creación de Promesas I

```
var verificar = clave => {
    return new Promise( (resolve, reject ) =>
{
        setTimeout( () => {
            if ( clave != 123 )
            {
                reject("Clave Errónea");
            }
            else
            {
                resolve("Ok");
            }
        }, 2000);
    });
}
```

```
verificar(123)
.then(data => {
    console.log(data); // Ok
})
.catch(error => {
    console.log(error); // Clave Errónea
});
```

Creación de Promesas II

```
verificar(123)
.then(data => {
   return data + '1';
})
.then(data => {
   return data + '2';
})
.then(data => {
   console.log(data);
})
.catch(error => {
   console.log(error);
});
```

Promesas

Métodos	Descripción
Promise.all(iterable)	Devuelve una de dos promesas: una que se cumple cuando todas las promesas en el argumento iterable han sido cumplidas, o una que se rechaza tan pronto como una de las promesas del argumento iterable es rechazada.
Promise.race(iterable)	Devuelve una promesa que se cumple o rechaza tan pronto como una de las promesas del iterable se cumple o rechaza, con el valor o razón de esa promesa.
Promise.reject(reason)	Devuelve un objeto Promise que es rechazado con la razón dada.
Promise.resolve(value)	Devuelve un objeto Promise que es resuelto con el valor dado.

Promesas

Lo bueno: Hemos recuperado en gran parte el control de flujo del programa de manera que el esquema de desarrollo de aplicaciones asíncronas basadas en promesas se parece algo más al estilo secuencial manteniendo su carácter no bloqueante.

- Recuperamos el return y la asignación.
- APIs más limpias sin métodos de callback (callback en cliente).
- Estructura del programa más similar a la programación secuencial.
- Razonamos con promesas como valores de futuro.

Lo malo: Aunque este modelo es altamente prometedor y supone grandes ventajas con respecto al de paso de continuaciones, no deja de tener detractores que consideran el uso de promesas un artefacto demasiado artificial e incómodo de tratar.

- No deja de ser necesario inyectar funciones manejadoras de éxito y error.
- Resulta difícil depurar hasta que las promesas no se han resuelto.
- Resulta más invasivo generar APIs que consumen y generan promesas.

Clausuras (closures)

- Los *closures* están compuestos por una función (llamémosla externa) que contiene otra función (llamémosla interna).
- La función interna tiene acceso a las variables de la función externa, creando un contexto que es persistente a través de distintas llamadas.
- Son funciones que llevan información asociada relativa al momento en que son invocadas.

Closures I

```
function contador()
{
    var i = 0;

    function closure()
    {
        i += 1;
        console.log( i );
    }

    return closure;
}

var miContador = contador();
```

```
miContador();

miContador();

miContador();

miContador();
```

Closures II

```
function persona()
{
    var nombre;

    return {
        getNombre: () => nombre,
        setNombre: nuevoNombre => {
            nombre = nuevoNombre;
        }
    }
}
```

```
var unaPersona = persona();

unaPersona.setNombre("Mario");
var nombre = unaPersona.getNombre();
```

Closures III

```
function btnGenerator( n ) {

        for ( var i = 1, btn; i <= n; i++ ) {

            btn = document.createElement("button");

            btn.innerHTML = "Btn " + i;

            btn.onclick = function () {
                alert(i);
            };

            document.body.appendChild(btn);
        }

    };
```

Contexto de una Función - this

- La función de la palabra clave **this** se comporta un poco diferente en Javascript en comparación con otros lenguajes.
- En general, el valor de **this** está determinado por cómo se llama a la función.
- El valor de **this** cambia según sobre qué objeto lo llamemos.
- El valor de **this** dentro de una función no depende de cómo se define esa función, sino de cómo se invoca.

This - Contexto Global

- En el contexto de ejecución global (fuera de cualquier función), **this** se refiere al objeto global, ya sea en modo estricto o no.

```
console.log( this === window ); // true

console.log( this.document === document ); // true

this.a = 37;

console.log(window.a); // 37
```

This - Contexto de la Función

- En modo estricto, el valor de **this** se mantiene en lo que está establecida al entrar en el contexto de ejecución.

```
function a(){

    console.log(this);
}

a();
```

```
function a(){
    'use strict';
    console.log(this);
}

a();
```

This - Como Método de un Objeto I

```
var nombre = "luke";

var apellido = "skywalker"

var o = {
   nombre: "Darth",
   apellido: "Vader",
   nombreCompleto: this.nombre + ', ' + this.apellido
}

console.log(o.nombreCompleto);
```

This - Como Método de un Objeto II

- Cuando una función es llamada como un método de un objeto, el **this** cambia por el método del objeto llamado.

```
var name = "Mario";

function myFunction() {

    console.log(this.name);
}

var objectA = {
    name: "Alice",
    myMethod: function () {
        console.log(this.name);
        }
};
```

```
var objectB = {
    name: "Bob",
    myMethod: myFunction
};

objectA.myMethod();

objectB.myMethod();

myFunction();
```

This - Como Método de un Objeto III

Regla: El objeto **this** pasado a una función es el objeto que está antes del punto que precede los paréntesis que invocan a la función.

```
Teniendo el método: objectA.myMethod()
```

- **objectA:** El objeto que contiene la función.
- **. (punto):** Separa el objeto de su propiedad (la función).
- **myMethod:** Nombre de la función.
- **() (paréntesis):** Ejecutan la función En éste caso vemos que antes del punto está objectA por lo que será objectA lo que se le pasará a la variable this del método myMethod.

* En los casos en los que haya más de un punto, **this** siempre es el objeto que está antes del último punto, es decir, el objeto que contiene la función.

This - Como Callback

```
var nombre = "Mario";
function esperarUnSegundo(callback) {
    setTimeout(function() {
        callback();
    }, 1000);
}
var alice = {
    nombre: "Alice",
    cansarse: function() {
      console.log(this.nombre + " se ha cansado de esperar");
    }
 };
 alice.cansarse();
 esperarUnSegundo(alice.cansarse);
```

This - Métodos de Function - apply y call

```
var alice = {
    nombre: "Alice",
    cansarse: function() {
      console.log(this.nombre + " se ha cansado de esperar");
    }
};

var miFuncion = alice.cansarse;

miFuncion();

miFuncion.call(alice, "arg1", "arg2");

miFuncion.apply(alice, ["arg1", "arg2"]);
```

This - Métodos de Function - bind

```
var alice = {
   nombre: "Alice",
   saludar: function() {
     console.log("Hola! Soy " +
this.nombre);
   }
};

function esperarUnSegundo(callback, context)
{
   setTimeout(function() {
     callback.call(context);
   }, 1000);
};
 esperarUnSegundo(alice.saludar, alice);
```

```
var myFunction = alice.saludar.bind(alice);

myFunction();
```

Módulos ES6

- Un módulo ES6 es un archivo que contiene código JS.
- En los módulos se pueden usar las sentencias *export* e *import*.
- En los módulos el codigo esta en modo estricto aun si no escribimos *“use strict”*.
- Todo lo declarado dentro del módulo es local al mismo.
- Para hacer el código visible se debe exportar.
- Los módulos son cacheados cuando se cargan e importaciones posteriores no recargan una nueva versión del módulo, sino que usan la versión cacheada anteriormente.
- Se puede exportar cualquier tipo de dato JS, funciones o clases.

Módulos ES6 - Exports con nombre.

```
export function sumar( a, b) {
      return a + b;
}
export function restar( a, b) {
      return a - b;
}
export function multiplicar( a, b) {
      return a * b;
}
o
export { sumar, restar, multiplicar}
```

```
/** Importar un solo método */
import { sumar } from './math.js';
/** Importar varios métodos */
import { sumar, restar } from './math.js';
/** Importar con alias */
import { sumar as s, restar as r } from './math.js';
/** Importar como namespace */
import * as math from './math.js';
```

Módulos ES6 - Exports por default.

```
export default function sumar( a, b)
{
      return a + b;
}
var math = {
  sumar: sumar,
  restar: restar,
  dividir: dividir
}
export default math;
```

*Solo puede puede haber un default.

```
/** Importar por defecto */
import sumar from './math.js';
```

Módulos ES6 - Módulos vs Scripts

	Scripts	Módulos
Elemento HTML	<script>	<script type="module">
Modo	No estricto	Estricto
Variables	Global	Local al módulo
Valor de *this*	window	undefined
Ejecución	Sincrónica	Asincrónica
Sentencia *import*	no	si
Extensión	.js	.js


JavaScript